

## Contenido

INTRODUCCIÓN .....	36
LOS FUNDAMENTOS DE MATERIAL DESIGN EN ANDROID .....	36
LAYOUTS .....	37
FrameLayout .....	38
LinearLayout .....	38
Relative Layout .....	38
TableLayout .....	38
GridLayout .....	39
PATRONES DE DISEÑO .....	39
FRAMELAYOUT .....	41
LINEARLAYOUT .....	42
RELATIVELAYOUT .....	45
TABLELAYOUT .....	46
TEMAS Y ESTILOS .....	49
Estilos .....	49
Herencia de estilos .....	49
Temas .....	50
Temas Y Estilos Del Sistema .....	53
Crear tu propio tema .....	53
Cambiar el fondo de nuestras actividades .....	54
Superponer la Action Bar .....	55
AÑADIR SCROLLVIEW .....	56
¿CÓMO AÑADIR LA TOOLBAR A NUESTRA APLICACIÓN? .....	57
¿CÓMO AÑADIR SCROLLING A LA TOOLBAR? .....	58
¿CÓMO AÑADIR EL EFECTO COLLAPSING A LA TOOLBAR? .....	60
¿CÓMO INCLUIR TOOLBAR EN OTRAS PARTES DE LA INTERFAZ? .....	61



# 3.

## Diseño de una aplicación Android. MaterialDesign

---

### INTRODUCCIÓN

Durante el pasado Google I/O 2014, la conferencia que da Google cada año, se nos presentaron muchas novedades siendo una de ellas esta guía de diseño. Material Design es un concepto, una filosofía, unas pautas enfocadas al diseño de aplicaciones utilizado en Android, pero también en la web y en cualquier plataforma.

Es un diseño donde la profundidad, las superficies, los bordes, las sombras y los colores juegan un papel principal.

Material Design es un diseño con una tipografía clara, casillas bien ordenadas, colores e imágenes llamativos para no perder el foco y un sentido del orden y la jerarquía muy marcado. Las luces y las sombras dan sensación de jerarquía.

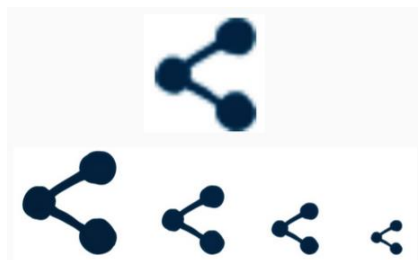
El movimiento es otro elemento clave, por ejemplo un objeto que parpadea significa que está llamando tu atención, un elemento que se expande es que se acaba de abrir. Todos estos movimientos se crean con una intención determinada.

No debemos pensar en Material Design como ese diseño destinado para las aplicaciones móviles de Android. De hecho, es multiplataforma. Tanto los smartphones, tablets, smartwatches o televisores pueden hacer uso de este diseño. También las páginas webs. Material Design ha sido creado pensando en todos los sistemas, no solo Android.

Material Design tiene sus propias normas para casi todos los detalles y se mantienen independientemente del tamaño de pantalla. Precisamente esa transversalidad es su punto fuerte.

### LOS FUNDAMENTOS DE MATERIAL DESIGN EN ANDROID

En primer lugar, es clave tener claro la unidad de medida que vamos a utilizar por delante del resto a la hora de ejecutar los diseños, porque debemos ser capaces de optimizar nuestro diseño para cualquier tipo de pantalla.



Para ello, debemos tener en cuenta que utilizaremos los famosos dpi, que son los píxeles independientes de densidad. Esta unidad presenta como ventaja el que podremos gestionar diferentes tipos de pantalla y sobre todo evitar tener los iconos distorsionados, como podemos ver en el anterior ejemplo. Para ello, deberemos definir bien nuestras imágenes en los diferentes tipos de pantallas (hdpi, xhdpi...).

Además, desde Android 5.0 (API 21), podemos utilizar incluso una particular imagen vectorizada, para evitar este tipo de problemas. Pero como es desde esta versión de Android, deberemos pensar si queremos ser o no compatibles con las anteriores, o cómo gestionarlo.

También debemos tener en cuenta que no todos nuestros drawables serán imágenes tal cual, sino que muchas de ellas irán definidas con ficheros XML, como por ejemplo serán aquellos que admitan diferentes estados. En este caso, no sólo necesitaremos una imagen en varias versiones, sino varias de ellas, y que las mismas tengan sentido visual.

Clarificado esto, deberemos tener en cuenta que a la hora de montar nuestro diseño, utilizaremos elementos View, los cuales presentan 3 elementos en común: content, padding y margin, donde el espacio que ocupa nuestra vista estará formado por el content y el padding (el cual será un espaciado que pertenece a la vista), y el margin hará alusión al espaciado entre elementos.

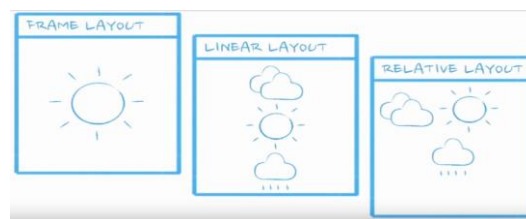


## LAYOUTS

Los *layouts* son elementos no visibles destinados a controlar la distribución, posición y dimensiones de los controles que se insertan en su interior. Estos componentes extienden la clase base *ViewGroup*.

La definición de la interfaz gráfica de una aplicación se definirá a través de archivos XML o a través de código Java de forma programática. Es mucho más eficiente y estructurado la primera de las opciones, no obstante, si en algunas ocasiones se requiere modificar alguna propiedad de los layouts o de una de sus vistas en tiempo real, usaremos la opción programática (por ejemplo hacer visible o invisible parte de una vista...)

Existen distintos tipos de *layouts*: ***FrameLayout***, ***LinearLayout***, ***RelativeLayout***, ***TableLayout***, ***GridLayout*** y ***ConstraintLayout***

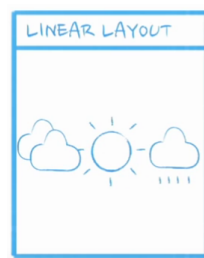


## FrameLayout

Es el más simple de todos los *layouts* de Android. Un *FrameLayout* coloca todos sus controles hijos alineados con su esquina superior izquierda, de forma que cada control quedará oculto por el control siguiente (a menos que éste último tenga transparencia). Por ello, suele utilizarse para mostrar un único control en su interior, a modo de contenedor sencillo para un sólo elemento sustituible, por ejemplo una imagen.

## LinearLayout

El siguiente *layout* Android en cuanto a nivel de complejidad es el *LinearLayout*., este *layout* apila uno tras otro todos sus elementos hijos de forma horizontal o vertical según se establezca su propiedad *android:orientation*.



Este sería la vista del layout con orientación horizontal.

## Relative Layout

Este *layout* permite especificar la posición de cada elemento de forma relativa a su elemento padre o a cualquier otro elemento incluido en el propio *layout*. De esta forma, al incluir un nuevo elemento X podremos indicar por ejemplo que debe colocarse *debajo del elemento Y*, y *alineado a la derecha del layout padre*.

*Todos estos elementos son combinables para poder diseñar la vista con la complejidad que se requiera.*

## TableLayout

Permite distribuir sus elementos hijos de forma tabular, definiendo las filas y columnas necesarias, y la posición de cada componente dentro de la tabla. La estructura de la tabla se define de forma similar a como se hace en HTML, es decir, indicando las filas que compondrán la tabla (objetos *TableRow*), y dentro de cada fila las columnas necesarias, con la salvedad de que no existe ningún objeto especial para definir una columna (algo así como un *TableColumn*) sino que directamente insertaremos los controles necesarios dentro del *TableRow* y cada componente insertado (que puede ser un control sencillo o incluso otro *ViewGroup*) corresponderá a una columna de la tabla.



## GridLayout

Este tipo de *layout* fue incluido a partir de la API 14 (Android 4.0) y sus características son similares al *TableLayout*, ya que se utiliza igualmente para distribuir los diferentes elementos de la interfaz de forma tabular, distribuidos en filas y columnas. La diferencia entre ellos estriba en la forma que tiene el *GridLayout* de colocar y distribuir sus elementos hijos en el espacio disponible. En este caso, a diferencia del *TableLayout* indicaremos el número de filas y columnas como propiedades del *layout*, mediante *android:rowCount* y *android:columnCount*.

## ConstraintLayout

ConstraintLayout está disponible como una biblioteca de soporte a partir del API 9 (Android 2.3) (Gingerbread). Permite crear diseños grandes y complejos con una jerarquía de vista plana (sin grupos de vista anidados). Es similar a *RelativeLayout*, pero más flexible y fácil de usar con el Editor de diseño de Android Studio.

## PATRONES DE DISEÑO

Toolbar

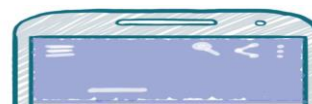


Está formada por un elemento de navegación (icono) con una funcionalidad determinada, el título de la misma y un menú de acciones, el situado más a la derecha se conoce como overflow y cuando pulsamos sobre él aparecen acciones adicionales que pueden ser seleccionadas.

App bar



App bar



Es un caso especial de toolbar que se sitúa en la parte superior de la pantalla. El título de la misma recoge el nombre de la aplicación o de la activity en uso. La appbar puede expandirse para disponer de más espacio para visualizar información adicional o una cabecera de imagen.

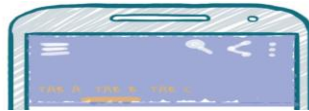
Para conseguir interacciones entre Views, el scroll de la AppBar o su expansión, Android nos ofrece el *CoordinatorLayout*,

Así si queremos el efecto de scroll sobre la AppBar (que esta desaparezca cuando nos desplazamos hacia arriba, o que aparezca cuando el desplazamiento es hacia abajo), combinaremos el *CoordinatorLayout* y la *ToolBar* con el *AppBarLayout*.

Si lo que deseamos es el efecto de colapsado y expansión de la AppBar utilizaremos el *collapsingToolbarLayout* en combinación con los anteriores.

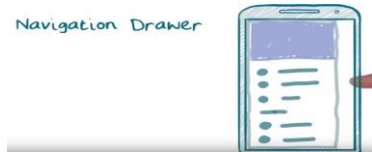


Tabs



Permiten la navegación entre diferentes secciones de la aplicación. Se agrupan junto a la AppBar y puedes cambiar de una a otra a través de gestos horizontales o desplazamientos izquierda y derecha.

Navigation Drawer



Es un panel que se desliza desde la izquierda de la pantalla y contiene el nivel más alto de las opciones de navegación. También puede visualizarse al presionar sobre el icono de la ActionBar. A su vez puede estar organizado en secciones.

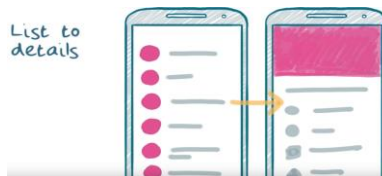
Scrolling and paging



Un ScrollView permite visualizar de forma cómoda la información cuando esta no cabe totalmente en la pantalla. Si la información puede visualizarse de forma completa no aparece el scroll y sino aparecerá de forma automática.

El ViewPager nos va a permitir disponer de varias páginas que serán accesibles con desplazamiento izquierda o derecha. Cuando implementamos Tabs, utilizamos ViewPagers para almacenar las vistas de cada uno de las opciones del Tab.

List to details



Es muy común el uso de este patrón de diseño en las aplicaciones Android. Cuando mostramos una lista de elementos y pulsamos sobre uno de ellos, aparece una segunda ventana donde se detalla la información del elemento pulsado.

Multipane



Es una concreción del patrón anterior, de forma que en dispositivos que dispongan de pantallas más grandes o dependiendo de si la pantalla la colocamos horizontal, visualizamos dos paneles o más en la pantalla, obteniendo una visualización optimizada de la vista y de su contenido.



## FRAMELAYOUT

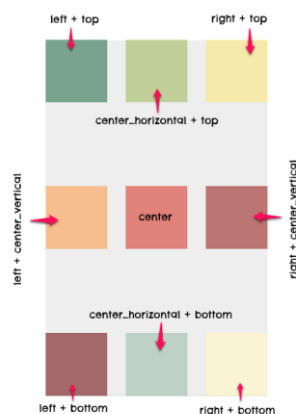
Recordad es el más simple de todos los *layouts* de Android. Un `FrameLayout` coloca todos sus controles hijos alineados con su esquina superior izquierda, de forma que cada control quedará oculto por el control siguiente (a menos que éste último tenga transparencia). Por ello, suele utilizarse para mostrar un único control en su interior, a modo de contenedor (placeholder) sencillo para un sólo elemento sustituible, por ejemplo una imagen.

Se utilizan las propiedades `android:layout_width` y `android:layout_height` para determinar las dimensiones del *layout*.

El *primero* especifica la anchura de la vista y el *segundo* especifica la altura de la vista

Es posible asignarles medidas absolutas definidas en **dps**, sin embargo *Google* recomienda hacerlo cuando sea estrictamente necesario, ya este tipo de medidas pueden afectar la UI en diferentes tipos de pantalla.

Para alinear el `FrameLayout` dentro contenedor padre se usa la propiedad `android:layout_gravity`.



El parámetro `gravity` se basa en las posiciones comunes de un `view` dentro del `layout`. Se describe con constantes de orientación:

- `top`: Indica la parte superior del `layout`.
- `left`: Indica la parte izquierda del `layout`.
- `right`: Se refiere a la parte derecha del `layout`.
- `bottom`: Representa el límite inferior del `layout`.
- `center_horizontal`: Centro horizontal del `layout`.
- `center_vertical`: Alineación al centro vertical del `layout`.
- `center`: Es la combinación de `center_horizontal` y `center_vertical`.

Es posible crear variaciones combinadas, como por ejemplo **`right | bottom`**.

Crear un nuevo proyecto y llamarlo `FrameLayout`. Abrir el archivo `activity_main.xml` y copiar la siguiente descripción de la vista:



```

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="android.pepe.pruebamaterialdesign.MainActivity">

    <FrameLayout
        android:layout_width="100dp"
        android:layout_height="50dp"
        android:layout_margin="10dp"
        android:background="#fff"
        android:elevation="4dp"
        android:layout_gravity="bottom|right">

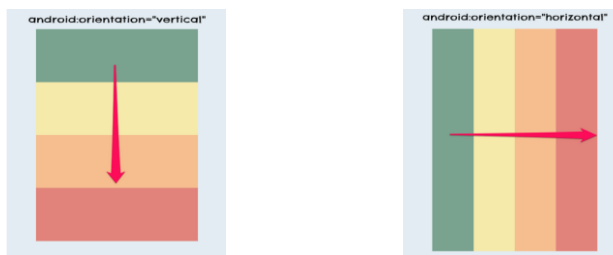
    </FrameLayout>
</FrameLayout>

```

Probar las distintas opciones para la propiedad `layout_gravity` y ver cómo cambia la apariencia de la vista.

## LINEARLAYOUT

El siguiente *layout* en cuanto a nivel de complejidad es el *LinearLayout*. Este *layout* apila uno tras otro, todos sus elementos hijos de forma horizontal o vertical según se establezca su propiedad `android:orientation`.



Crear un proyecto nuevo y llamarlo *LinearLayout*, editar el archivo `activity_main`:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.example.josebalmasedadelalamo.linearlayout.MainActivity">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="Salir" />

</LinearLayout>

```

igual que en un *FrameLayout*, los elementos contenidos en un *LinearLayout* pueden establecer sus propiedades `android:layout_width` y `android:layout_height` para determinar sus dimensiones dentro del *layout*.

Aunque aún no lo hayamos visto, hemos incluido un nuevo view llamado *Button*.

- `wrap_content`: Ajusta el tamaño al espacio mínimo que requiere el view. En el siguiente ejemplo se ve como un botón ajusta su ancho y alto a cantidad necesaria para envolver el texto interior. Y el resultado sería:





- `match_parent`: Ajusta el tamaño a las dimensiones máximas que el padre le permita. La siguiente ilustración muestra el mismo botón anterior, solo que asignado `match_parent` a su parámetro `layout_width`.

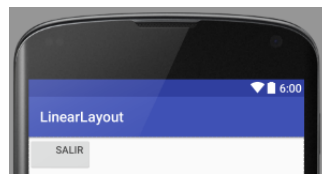
```
android:layout_width="match_parent"
android:layout_height="wrap_content"
```

Y el resultado sería:



Una particularidad del parámetro **android:layout\_gravity** cuando se utiliza en un **LinearLayout** (y que no se daba en el **FrameLayout**) es que las opciones de alineamiento que afectan a la dirección en la que se orienta el **LinearLayout** se ignoran.

El atributo `gravity` de una vista alinea los elementos situados dentro del contenedor según el valor asignado (derecha, izquierda,...). Por ejemplo dentro de una vista como `Button`, alinea el texto dentro del botón. Los valores que puede tomar esta propiedad son iguales que los de `layout_gravity`.



Adicionalmente podemos definir también otro parámetro llamado `android:layout_weight` que permite especificar el tamaño de las vistas existentes en el layout. Para comprenderlo insertamos tres botones:

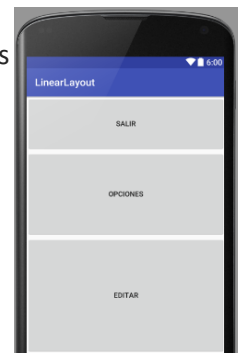
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.example.josebalmasedadelalamo.linearlayout.MainActivity">

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Salir"
        android:layout_weight="1"/>

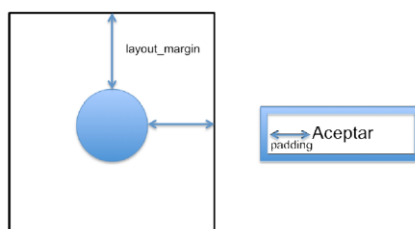
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Opciones"
        android:layout_weight="2"/>

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Editar"
        android:layout_weight="3"/>
</LinearLayout>
```

Matemáticamente, el espacio disponible total sería la suma de las alturas (6), por lo que 3 representa el 50%, 2 el 33,33% y 1 el 16,66%.



En Android, el espaciado se especifica mediante los atributos *padding* y *layout\_margin*. El atributo *layout\_margin* espacia la vista con respecto a su contenedor u otras vistas, mientras que *padding* espacia el contenido de una vista respecto a los bordes de la vista. Dicho de otra forma, *layout\_margin* especifica el espaciado fuera de los bordes de la vista mientras que *padding* lo hace dentro de los bordes de la vista, como se muestra en el siguiente diagrama:



Se puede ser más específico aún:

- *android:padding*: especifica el espacio vacío entre el contenido de un elemento y sus cuatro lados.
- *android:paddingTop*: especifica el espacio vacío entre el contenido de un elemento y su lado superior.
- *android:paddingBottom*: especifica el espacio vacío entre el contenido de un elemento y su lado inferior.
- *android:paddingLeft*: especifica el espacio vacío entre el contenido de un elemento y su lado izquierdo.
- *android:paddingRight*: especifica el espacio vacío entre el contenido de un elemento y su lado derecho.

El espacio entre una vista y su contenedor se especifica mediante los siguientes atributos:

- *android:layout\_margin*: especifica el espacio entre una vista y las vistas o el contenedor por sus cuatro lados.
- *android:layout\_marginTop*: especifica el espacio entre el lado superior de una vista y otro elemento o el contenedor.
- *android:layout\_marginBottom*: especifica el espacio entre el lado inferior de una vista y otro elemento o el contenedor.
- *android:layout\_marginLeft*: especifica el espacio entre el lado izquierdo de una vista y otro elemento o el contenedor.
- *android:layout\_marginRight*: especifica el espacio entre el lado derecho de una vista y otra vista o el contenedor.



El siguiente ejemplo ilustra estas diferencias:

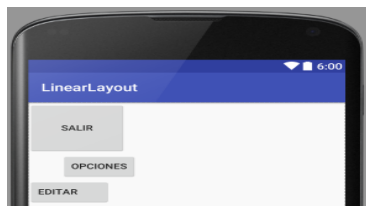
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.example.josebalmasedadelalamo.linearlayout.MainActivity">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Salir"
        android:padding="40dp"/>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Opciones"
        android:layout_marginLeft="40dp"/>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Editar"
        android:paddingRight="40dp"/>

</LinearLayout>
```



El aspecto que se consigue:

## RELATIVELAYOUT

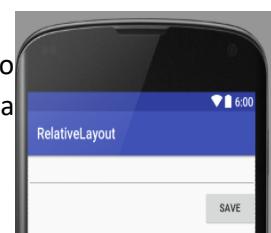
Este *layout* permite especificar la posición de cada elemento de forma relativa a su elemento padre o a cualquier otro elemento incluido en el propio *layout*. De esta forma, al incluir un nuevo elemento X podremos indicar por ejemplo que debe colocarse *debajo del elemento Y*, y *alineado a la derecha del layout padre*. Veamos esto en el ejemplo siguiente, definir un nuevo proyecto y llamarlo RelativeLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.josebalmasedadelalamo.relativelayout.MainActivity">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Save"
        android:id="@+id/button"
        android:layout_alignParentRight="true"
        android:layout_below="@+id/editText"/>

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/editText"
        android:layout_alignParentTop="true"
        android:layout_toStartOf="@id/button"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"/>

</RelativeLayout>
```



En el ejemplo, el botón BtnAceptar se colocará debajo del cuadro de texto TxtNombre (`android:layout_below="@id/TxtNombre"`) y alineado a la derecha del layout padre (`android:layout_alignParentRight="true"`).



Al igual que estas propiedades, en un `RelativeLayout` tendremos un sinnúmero de propiedades para colocar cada control justo donde queramos. Veamos las principales (creo que sus propios nombres explican perfectamente la función de cada una):

Tipo	Propiedades
Posición relativa a otro control	<code>android:layout_above</code> <code>android:layout_below</code> <code>android:layout_toLeftOf</code> <code>android:layout_toRightOf</code> <code>android:layout_alignLeft</code> <code>android:layout_alignRight</code> <code>android:layout_alignTop</code> <code>android:layout_alignBottom</code> <code>android:layout_alignBaseline</code>
Posición relativa al layout padre	<code>android:layout_alignParentLeft</code> <code>android:layout_alignParentRight</code> <code>android:layout_alignParentTop</code> <code>android:layout_alignParentBottom</code> <code>android:layout_centerHorizontal</code> <code>android:layout_centerVertical</code> <code>android:layout_centerInParent</code>
Opciones de margen (también disponibles para el resto de layouts)	<code>android:layout_margin</code> <code>android:layout_marginBottom</code> <code>android:layout_marginTop</code> <code>android:layout_marginLeft</code> <code>android:layout_marginRight</code>
Opciones de espaciado o padding (también disponibles para el resto de layouts)	<code>android:padding</code> <code>android:paddingBottom</code> <code>android:paddingTop</code> <code>android:paddingLeft</code> <code>android:paddingRight</code>

## TABLELAYOUT

**TableLayout** permite distribuir sus elementos hijos de forma tabular, definiendo las filas y columnas necesarias, y la posición de cada componente dentro de la tabla. Abrir un nuevo proyecto con el nombre *TableLayout*.

La estructura de la tabla se define de forma similar a como se hace en HTML, es decir, indicando las filas que compondrán la tabla (objetos *TableRow*), y dentro de cada fila las columnas necesarias, con la salvedad de que no existe ningún objeto especial para definir una columna (algo así como un *TableColumn*) sino que directamente insertaremos los controles necesarios dentro del *TableRow* y cada componente insertado (que puede ser un control sencillo o incluso otro *ViewGroup*) corresponderá a una columna de la tabla. De esta forma, el número final de filas de la tabla se corresponderá con el número de elementos *TableRow* insertados, y el número total de columnas quedará determinado por el número de componentes de la fila que más componentes contenga.



```

<TableLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TableRow>
        <TextView android:text="Celda 1.1" />
        <TextView android:text="Celda 1.2" />
        <TextView android:text="Celda 1.3" />
    </TableRow>

    <TableRow>
        <TextView android:text="Celda 2.1" />
        <TextView android:text="Celda 2.2" />
        <TextView android:text="Celda 2.3" />
    </TableRow>

    <TableRow>
        <TextView android:text="Celda 3.1"
            android:layout_span="2" />
        <TextView android:text="Celda 3.2" />
    </TableRow>
</TableLayout>

```

Por norma general, el ancho de cada columna se corresponderá con el ancho del mayor componente de dicha columna, pero existen una serie de propiedades del **TableLayout** que nos ayudarán a modificar este comportamiento:

- *android:layout\_colum*. Indicará la columna a la que pertenece una celda dentro del TableRow.
- *android:stretchColumns*. Indicará las columnas que pueden expandir para absorber el espacio libre dejado por las demás columnas a la derecha de la pantalla.
- *android:shrinkColumns*. Indicará las columnas que se pueden contraer para dejar espacio al resto de columnas que se puedan salir por la derecha de la pantalla.
- *android:collapseColumns*. Indicará las columnas de la tabla que se quieren ocultar completamente.

Todas estas propiedades del *TableLayout* pueden recibir una lista de índices de columnas separados por comas (ejemplo: *android:stretchColumns="1,2,3"*) o un asterisco para indicar que debe aplicar a todas las columnas (ejemplo: *android:stretchColumns="\*"*).

Otra característica importante es la posibilidad de que una celda determinada pueda ocupar el espacio de varias columnas de la tabla (análogo al atributo *colspan* de HTML). Esto se indicará mediante la propiedad **android:layout\_span**, pero esta vez aplicado al componente concreto que deberá tomar dicho espacio (por ejemplo como atributo del *TextView*).

Para más información: <https://columna80.wordpress.com/2012/11/05/diseos-android-bsicos-tablelayout/>

**GridLayout** este tipo de *layout* fue incluido a partir de la API 14 (Android 4.0) y sus características son similares al *TableLayout*, ya que se utiliza igualmente para distribuir los diferentes elementos de la interfaz de forma tabular, distribuidos en filas y columnas. La diferencia entre ellos estriba en la forma que tiene el *GridLayout* de colocar y distribuir sus elementos hijos en el espacio disponible. En este caso, a diferencia del *TableLayout* indicaremos el número de filas y columnas como propiedades del *layout*, mediante *android:rowCount* y *android:columnCount*. Con estos datos ya no es necesario ningún tipo de elemento para indicar las filas, como hacíamos con el elemento *TableRow* del *TableLayout*, sino que los diferentes elementos hijos se irán colocando ordenadamente por filas o columnas (dependiendo de la propiedad *android:orientation*) hasta completar el número de filas o columnas indicadas en los atributos anteriores. Adicionalmente, igual que en el caso anterior, también tendremos disponibles las



propiedades ***android:layout\_rowSpan*** y ***android:layout\_columnSpan*** para conseguir que una celda ocupe el lugar de varias filas o columnas. Crear archivo *activity\_main\_v2* en el proyecto *tableLayout*. Con todo esto en cuenta, para conseguir una distribución equivalente a la del ejemplo anterior del *TableLayout*, necesitaríamos escribir un código como el siguiente:

```
<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:rowCount="2"
    android:columnCount="3"
    android:orientation="horizontal" >

    <TextView android:text="Celda 1.1" />
    <TextView android:text="Celda 1.2" />
    <TextView android:text="Celda 1.3" />

    <TextView android:text="Celda 2.1" />
    <TextView android:text="Celda 2.2" />
    <TextView android:text="Celda 2.3" />

    <TextView android:text="Celda 3.1"
        android:layout_columnSpan="2" />

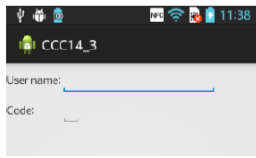
    <TextView android:text="Celda 3.2" />

</GridLayout>
```

Aunque en el ejemplo veamos el atributo ***android:rowCount*** a 2, como el ***columnCount*** está a 3, hace la agrupación en 3 columnas y los 2 *TextView* que quedan los considera otra fila.

También es posible especificar cada *View* en que fila y columna van a colocarse, esto se hace con los atributos ***android:layout\_row*** y ***android:layout\_column***. Veamos un ejemplo:

```
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_row="0"
        android:layout_column="0"
        android:text="@string/user_name" />
    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_row="0"
        android:layout_column="1"
        android:inputType="textPersonName"
        android:minWidth="200dp"/>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_row="1"
        android:layout_column="0"
        android:text="@string/code" />
    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_row="1"
        android:layout_column="1"
        android:inputType="textPassword"/>
</GridLayout>
```



La propiedad ***android:inputType*** indica el tipo de contenido que se va a introducir en el cuadro de texto, como por ejemplo una dirección de correo electrónico (***textEmailAddress***), un número genérico (***number***), un número de teléfono (***phone***), etc. El valor que establezcamos para esta propiedad tendrá además efecto en el tipo de teclado que mostrará Android para editar dicho campo.

## CONSTRAINTLAYOUT

***ConstraintLayout*** en este contenedor para definir la posición de una vista, se le debe agregar al menos una restricción horizontal y una vertical a la vista. Cada restricción representa una conexión o alineación con otra vista, el diseño principal o una línea invisible. Cada restricción define la posición de la vista a lo largo del eje vertical u horizontal; por lo que cada vista debe tener un mínimo de una restricción para cada eje, pero a menudo son necesarias más. Con este layout se trabaja principalmente a nivel de diseño con la herramienta del editor, por lo que la mejor manera de entenderlo es visitando el enlace <https://developer.android.com/training/constraint-layout/>

### Ejercicio Propuesto Layouts



## TEMAS Y ESTILOS

Los estilos y temas son una herramienta que facilita Android para ayudarnos en el diseño de la aplicación. Un estilo es una colección de propiedades que permite especificar el aspecto y formato de una vista o una ventana. Un tema es lo mismo que un estilo, pero aplicado a una actividad al completo, no sólo a una vista. Los estilos en Android siguen una filosofía similar a los CSS en el diseño web que nos permite separar el diseño de la aplicación del contenido. El siguiente ejemplo nos servirá para asentar el concepto:

### *Estilos*

La forma más práctica de crear estilos, es generarlos en un archivo de recursos que nos permita reusar el código. Para ello debemos crear un nuevo archivo XML que se albergue en la carpeta de recursos **res/values/**. Donde usaremos como nodo padre para los recursos la etiqueta `<resources>` y para definir un estilo usaremos el elemento `<style>`. Se asignará un nombre único a través de su atributo `name`. Para definir las reglas que lo componen crearemos elementos `<item>` en su interior, detallando el nombre del atributo a modificar y su respectivo valor.

Veamos un ejemplo, crea un proyecto llamado `ejemploEstilos` y añade las siguientes líneas en un archivo que te crees nuevo llamado `res/values/misestilos`:

```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
    <style name="buttonStyle">
        <item name="android:layout_width">wrap_content</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#AEC6CF</item>
    </style>
</resources>
```

Si deseáramos implementar este estilo en un botón dentro de un layout, entonces referenciamos un acceso a los recursos de estilos con la convención `@style/nombreEstilo`, para ello añade en el archivo `activity_main.xml` el siguiente código, dentro de la etiqueta padre.

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical">

    <Button
        style="@style/buttonStyle"
        android:text="Cliqueame" />

</LinearLayout>
```

### *Herencia de estilos*

El elemento `<style>` también puede heredar propiedades de otro estilo a través de su atributo `parent`. Esta relación permite copiar las reglas del estilo padre y sobrescribir o añadir propiedades. Veamos un ejemplo:



```
<style name="buttonStyle" parent="@style/parentStyle">
```

Como ves, referenciamos a otro estilo llamado parentStyle.

Cabe aclarar que siempre que creas un nuevo proyecto en Android Studio, el archivo styles.xml es autogenerado con una estructura similar a esta:

```
<resources>
    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>
</resources>
```

El estilo hereda sus propiedades del estilo padre Theme.AppCompat.Light.DarkActionBar. En el siguiente enlace aparecen todos los estilos que proporciona Android: <http://developer.android.com/reference/android/R.style.html>

Pero si lo que queremos es heredar de un estilo ya definido por nosotros no hay que utilizar parent. En el siguiente código de ejemplo podemos ver cómo hacer un estilo que herede de nuestro estilo creado "ButtonStyle", y modificar el tamaño de la letra para que ahora sea más pequeña.

```
<style name="buttonStyle.Pequeña">
    <item name="android:textSize">15px</item>
</style>
```

De esta forma podemos encadenar herencias, como por ejemplo, un estilo que herede de "Estilo.Pequeña" y que modifique el color a verde

```
<style name="buttonStyle.Pequeña.Verde">
    <item name="android:textColor">#00ff00</item>
</style>
```

Haciendo referencia a este estilo en la activity\_main, de la siguiente manera:

```
<Button
    style="@style/buttonStyle.pequeña.verde"
    android:text="Cliqueame" />
```

## Temas

Un tema es un estilo genérico que se asigna a una aplicación completa o actividad. Esto permite que todos los componentes sigan un mismo patrón de diseño y personalización para mantener consistencia en la UI. Si deseamos añadir un tema a una aplicación debemos dirigirnos al archivo AndroidManifest.xml y agregar al elemento <application> el atributo theme con la referencia del tema solicitado. Veamos:

```
<application android:theme="@style/MiTema">
```

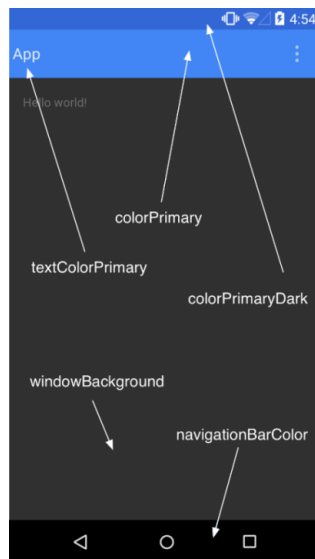
Si fuese una actividad entonces haríamos exactamente lo mismo:

```
<activity android:theme="@style/TemaActividad">
```

Si te fijas en el Android Manifest, se asigna por defecto un tema que se encuentra en el namespace del sistema con la referencia @android/style/AppTheme. Alguno de los elementos configurables del tema vienen definidos siguiendo el siguiente esquema:







Crear un proyecto con el nombre pruebaMaterialDesign. Por defecto se crea el archivo `res/values/colors.xml` con el siguiente contenido:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
</resources>
```

Y el archivo `res/values/styles.xml` como sigue:

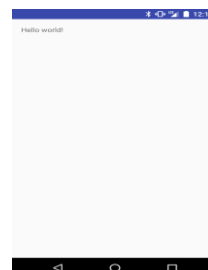
```
<resources>

    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>
</resources>
```

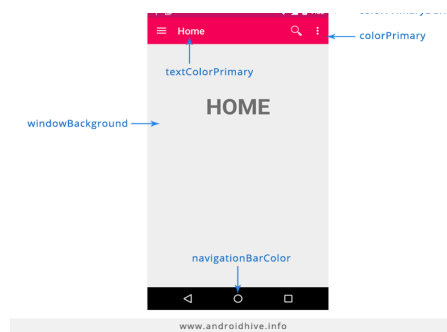
Como se puede ver, se utiliza un tema con `NoActionBar` porque queremos utilizar la `AppBar`.

Podemos cambiar cualquiera de las características de color añadiendo ítems o modificando los existentes e ir personalizando el aspecto de nuestra aplicación.

La aplicación tendrá el siguiente aspecto:



Vamos a personalizar nuestra aplicación para que tenga el aspecto siguiente:



Para ello modificaremos los siguientes archivos:

```
colors.xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#F50057</color>
    <color name="colorPrimaryDark">#C51162</color>
    <color name="textColorPrimary">#FFFFFF</color>
    <color name="windowBackground">#FFFFFF</color>
    <color name="navigationBarColor">#000000</color>
    <color name="colorAccent">#FF80AB</color>
</resources>
```

```
styles.xml
<resources>

    <style name="MyMaterialTheme" parent="MyMaterialTheme.Base">
    </style>

    <style name="MyMaterialTheme.Base" parent="Theme.AppCompat.Light.DarkActionBar">
        <item name="windowNoTitle">true</item>
        <item name="windowActionBar">false</item>
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>
</resources>
```

Ahora hay que modificar el archivo de manifiesto para aplicar este tema a nuestra aplicación:



```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="android.pepe.pruebamaterialdesign">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="pruebaMaterialDesign"
        android:supportRtl="true"
        android:theme="@style/MyMaterialTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>

```

Eso nos lleva a nuestro siguiente apartado...

## *Temas Y Estilos Del Sistema*

Android trae por defecto estilos y temas para todas sus aplicaciones y entorno. Estas reglas de estilos son guardadas en un archivo llamado `styles.xml` y los temas en `themes.xml`. Ambos contienen definiciones establecidas por el equipo desarrollador de Android creadas a su gusto y medida.

Antes de la versión 11 se usaba un tema por defecto llamado `Theme.Light`, pero para las versiones recientes se diseñaron los temas `Theme.AppCompat` (Estilo oscuro) y el `Theme.AppCompat.Light` (Estilo claro).

De ellos descienden muchas variantes, como por ejemplo el tema `Theme.AppCompat.Light.DarkActionBar`.



Figure 1. Dark material theme

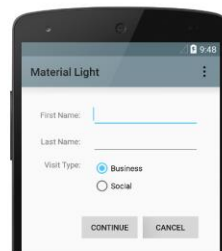


Figure 2. Light material theme

## *Crear tu propio tema*

Para facilitar la personalización de un tema nuevo es recomendable extender las propiedades de los temas que Android contiene. Esto nos permitirá ahorrarnos tiempo en definición y escritura, por lo que solo se implementan las reglas que deseamos modificar en particular.

Supongamos que deseamos usar el tema `AppCompat.Light` en nuestra aplicación, pero con el formato de texto *itálico*. Para conseguir este resultado aplicamos el mismo procedimiento que hicimos con los estilos, donde nuestro tema heredará la mayoría de características del tema



padre y solo tendremos que editar el atributo `android:textStyle`. No olvidar referenciar al tema `Italic`, en el atributo correspondiente del manifest, como hemos visto anteriormente.

```
<style name="Italic" parent="Theme.AppCompat.Light">
    <item name="android:textStyle">italic</item>
</style>
```

## *Cambiar el fondo de nuestras actividades*

Es normal que deseemos cambiar el aspecto con que se proyecta una actividad en su interior por un color llamativo o una imagen de fondo. Para hacerlo, acudimos a la propiedad `windowBackground`. Los atributos que empiezan por el prefijo `window` no son aplicables a un `view` en concreto. Ellos se aplican a una `app` o actividad como si se tratase de un todo o un solo objeto.

Este atributo recibe por referencia un color sólido, una forma o una imagen de nuestros recursos. Normalmente los colores se deben declarar como ítems `<color>`, cuyo valor es un número hexadecimal. Tendremos que referenciar al nuevo tema en el Manifest.

```
<style name="Fondo" parent="AppTheme">
    <item name="android:windowBackground">@android:color/holo_blue_light</item>
</style>
```

En este caso usamos un color predefinido por el sistema. El resultado sería este:



Si deseas usar tu propio color deberás declarar tu ítem `<color>` en el archivo de recursos `colors.xml` y asignarlo al nuevo estilo:

```
<color name="yellowPastel">#FDFD96</color>

<style name="ColorPropio" parent="AppTheme">
    <item name="android:colorBackground">@color/yellowPastel</item>
    <item name="android:windowBackground">@color/yellowPastel</item>
</style>
```

Ahora tendríamos el siguiente fondo:



Para usar una imagen como fondo de la aplicación o de la actividad, simplemente haremos una referencia a la carpeta `drawable`, donde tendremos guardada la imagen.

```
<style name="ImagenPropia" parent="AppTheme">
    <item name="android:windowBackground">@drawable/fondo</item>
</style>
```



La siguiente ilustración muestra una imagen de fondo:



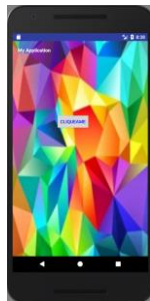
## Superponer la ActionBar

En ocasiones deseamos que nuestra ActionBar no interfiera en la visualización de nuestra actividad. Una de las maneras sería sobreponerla en el fondo de la actividad. Esto nos permitirá contrastarla de forma eficaz. Para ello debemos configurar un estilo para el TextView que se encuentra dentro de la ActionBar, una serie de propiedades a la propia ActionBar y para finalizar, el estilo que consigue la transparencia de esta. Pasamos a mostrar el código:

```
<style name="ImagenPropia" parent="AppTheme">
    <item name="android:windowBackground">@drawable/fondo</item>
    <item name="android:windowContentOverlay">@null</item>

    <item name="android:windowActionBarOverlay">true</item>
    <item name="android:actionBarStyle">@style/MyActionBar</item>
    <!-- Support library compatibility -->
    <item name="windowActionBarOverlay">true</item>
    <item name="actionBarStyle">@style/MyActionBar</item>
</style>
<!-- ActionBar styles -->
<style name="MyActionBar" parent="@style/Widget.AppCompat.Light.ActionBar.Solid.Inverse">
    <item name="android:background">@android:color/transparent</item>
    <!-- Support library compatibility -->
    <item name="background">@android:color/transparent</item>
    <!-- Text style ActionBar -->
    <item name="titleTextStyle">@style/TitleTextStyle</item>
</style>
<style name="TitleTextStyle" parent="@android:style/Widget.TextView">
    <item name="android:textStyle">bold</item>
    <item name="android:textSize">15dp</item>
    <item name="android:textColor">@android:color/white</item>
</style>
```

La anterior descripción produciría un efecto similar al siguiente:



Si deseas que la ActionBar se vea translúcida, será un pequeño cambio, deberás aplicar un color transparente al background. Para ello nos podemos definir el color en el archivo color.xml y posteriormente aplicarlo a los atributos correspondientes.

```
<color name="translucido">#4000</color>
```



```

<style name="MyActionBar" parent="@style/Widget.AppCompat.Light.ActionBar.Solid.Inverse">
    <item name="android:background">@color/translucido</item>
    <!--Support library compatibility -->
    <item name="background">@color/translucido</item>
    <!-- Text style ActionBar -->
    <item name="titleTextStyle">@style/TitleTextStyle</item>
</style>

```

El resultado:



## SCROLLVIEW Y ELEVATION

Otro de los conceptos que aporta MaterialDesign es el de hacer relevante algún elemento resaltando la superficie de este. Para ello podemos utilizar **la propiedad elevation**. Veamos un ejemplo sencillo donde combinamos dos tipos de layouts y la propiedad elevation. Abrir el proyecto llamado pruebaMaterialDesign y editar el archivo activity\_main:

```

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="16dp"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    tools:context="android.pepe.pruebamaterialdesign.MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">

        <FrameLayout
            android:layout_width="match_parent"
            android:layout_height="200dp"
            android:layout_margin="10dp"
            android:background="#fff"
            android:elevation="4dp">

        </FrameLayout>
    </LinearLayout>
</FrameLayout>

```



Para ver el efecto insertar ahora otros dos FrameLayout con elevation 8dp y 16dp (activity\_main\_v2).

Otro concepto muy útil a la hora de crear vistas, es el de poder movernos por la pantalla para poder visualizar los contenidos que no caben. Para ello podemos introducir un elemento de tipo ScrollView, de este tipo de elementos nos encontramos con el ScrollView (vertical) y el



HorizontalScrollView (horizontal). Para ver este ejemplo vamos a trabajar sobre el proyecto pruebaMaterialDesig. Vamos a modificar el archivo activity\_main.xml que contiene el diseño de nuestra interfaz gráfica (activity\_main\_v3).

```
<ScrollView
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">

        <FrameLayout
            android:layout_width="match_parent"
            android:layout_height="200dp"
            android:layout_margin="10dp"
            android:background="#fff"
            android:elevation="4dp">

        </FrameLayout>

        <FrameLayout
            android:layout_width="match_parent"
            android:layout_height="200dp"
            android:layout_margin="10dp"
            android:background="#fff"
            android:elevation="8dp">

        </FrameLayout>

        <FrameLayout
            android:layout_width="match_parent"
            android:layout_height="200dp"
            android:layout_margin="10dp"
            android:background="#fff"
            android:elevation="12dp">

        </FrameLayout>
    </LinearLayout>
</ScrollView>
```

Es importante tener en cuenta que el scrollView solo puede contener un elemento o hijo.

## ¿CÓMO AÑADIR LA TOOLBAR A NUESTRA APLICACIÓN?

Una de las novedades de Material Design, ha sido la aparición de la Toolbar que pretende ser un reemplazo de la ActionBar más potente y flexible. Mientras que la ActionBar es un elemento del sistema que se muestra en una Activity si se hereda de un tema que la incluya, Toolbar es simplemente un widget que aporta grandes ventajas:

- Puede ubicarse en cualquier lugar.
- Se puede ubicar más de una en la misma pantalla.
- Es un ViewGroup por lo que se pueden incluir dentro la Toolbar otros widgets.
- Fácilmente adaptable al tamaño de pantalla.
- Sigue proporcionado la funcionalidad de la ActionBar facilitando por tanto su adopción.

Debido a que la Toolbar reemplaza a la antigua ActionBar, debes deshabilitarla con el estilo Theme.AppCompat.NoActionBar ó añadiendo los atributos windowActionBar y windowNoTitle.



Esto ya lo habíamos configurado cuando trabajamos con el proyecto pruebaMaterialDesign. Ahora lo único que tenemos que hacer es colocar o definir un widget del tipo Toolbar en nuestro archivo de diseño con la etiqueta `<android.support.v7.widget.Toolbar>`. Llamaremos al archivo `activity_main_v4`.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        android:theme="@style/ThemeOverlay.AppCompat.Dark"/>

    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <ScrollView
            android:layout_width="match_parent"
            android:layout_height="match_parent">

            <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:orientation="vertical">

                <FrameLayout
                    android:layout_width="match_parent"
                    android:layout_height="200dp"
                    android:layout_margin="10dp"
                    android:background="#fff"
                    android:elevation="4dp">

                </FrameLayout>

            ...
        </ScrollView>
    </FrameLayout>
</LinearLayout>
```

Hemos cambiado el contenedor principal a `LinearLayout`, con una configuración de la propiedad `orientation` vertical. El `LinearLayout` contiene la `toolbar` y un `FrameLayout` con el `scrollView`.

## ¿CÓMO AÑADIR SCROLLING A LA TOOLBAR?

Una de las cosas más sorprendentes acerca de la biblioteca de diseño que estamos analizando es que podemos crear interfaces de usuario animadas con una simple configuración en el archivo XML. No requiere ningún código, por lo que su implementación es un proceso muy sencillo.

Es imprescindible el utilizar un `CoordinatorLayout` para coordinar los elementos que necesitan animación, este tipo de layout tiene que ser el nivel más alto en la jerarquía de la interfaz, ya que es el encargado de las efectuar las animaciones entre las vistas que se definen en él.

Otro de los elementos que necesitamos para conseguir el efecto de scrolling de la toolbar es la `AppBarLayout`, este elemento permite la inclusión de la toolbar y de más elementos que se pueden añadir a ella, como pestañas, imágenes, etc.





Es obligatorio que haya un elemento con scroll en la interfaz y deberá ser marcado o indicado con el atributo `app:layout_behavior="@string/appbar_scrolling_view_behavior"`.

Veamos cómo implementar este efecto en la aplicación que tenemos abierta, llamaremos al archivo `activity_main_v5`, en él incluiremos un elemento nuevo llamado `NestedScrollView`, que permite también hacer scroll sobre un elemento, pero que además es compatible para realizar el scroll con la `appbar` (otro elemento compatible es el `RecyclerView`, que veremos en temas posteriores). Vamos a probar el comportamiento diferente, si aplicamos `app:layout_behavior="@string/appbar_scrolling_view_behavior"` sobre el `TextView` o sobre el `NestedScrollView`.

```
<android.support.design.widget.CoordinatorLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MainActivity">

    <android.support.design.widget.AppBarLayout
        android:id="@+id/app_bar_layout"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:gravity="center"
        android:theme="@style/ThemeOverlay.AppCompat.Dark">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            android:theme="@style/ThemeOverlay.AppCompat.Dark"
            app:layout_scrollFlags="scroll|enterAlways"/>
    </android.support.design.widget.AppBarLayout>

    <android.support.v4.widget.NestedScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            android:text="@string/hello_world"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textStyle="bold"
            android:textSize="18dp"
            app:layout_behavior="@string/appbar_scrolling_view_behavior"/>
    </android.support.v4.widget.NestedScrollView>
</android.support.design.widget.CoordinatorLayout>
```

El atributo `app:layout_scrollFlags` determina cual va a ser el comportamiento de la view (en nuestro caso la `toolbar`) a través de los siguientes valores:

- `scroll`: indica que la view desaparecerá al desplazar el contenido.
- `enterAlways`: vuelve visible la view ante cualquier signo de scrolling.
- `enterAlwaysCollapsed`: vuelve visible al view solo si se mantiene el scroll en la parte superior del contenido.
- `exitUntilCollapsed`: desaparece la view hasta que sus dimensiones superen la altura mínima.



## ¿CÓMO AÑADIR EL EFECTO COLLAPSING A LA TOOLBAR?

Para implementar esta característica se usa el `CollapsingToolbarLayout`. Un layout especial que envuelve a la `Toolbar` para controlar las reacciones de expansión y contracción de los elementos que se encuentran dentro de un `AppBarLayout`. Veamos el esquema general del archivo `activity_main_v6`.

El layout contenedor debe ser de nuevo un `CoordinatorLayout`.

El segundo elemento a integrar será el `AppBarLayout`. Este elemento contendrá, en nuestro caso, la `Toolbar` y una imagen de fondo para el `AppBarLayout`. Si queremos introducir los efectos de scroll sobre la `Toolbar`, será necesario encapsular la imagen y la `Toolbar` dentro de otro elemento que se conoce como `CollapsingToolbarLayout`.

```
<android.support.design.widget.CoordinatorLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MainActivity">

    <android.support.design.widget.AppBarLayout
        android:id="@+id/app_bar_layout"
        android:layout_height="250dp"
        android:layout_width="match_parent"
        android:gravity="center"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">

        <android.support.design.widget.CollapsingToolbarLayout
            android:id="@+id/collapsing_toolbar"
            android:layout_height="match_parent"
            android:layout_width="match_parent"
            app:contentScrim="?attr/colorPrimary"
            app:layout_scrollFlags="scroll|exitUntilCollapsed">

            <ImageView
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:scaleType="centerCrop"
                android:src="@drawable/image"
                app:layout_collapseMode="parallax"/>

            <android.support.v7.widget.Toolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                app:layout_collapseMode="pin"/>

        </android.support.design.widget.CollapsingToolbarLayout>

    </android.support.design.widget.AppBarLayout>

</android.support.design.widget.CoordinatorLayout>
```

Existen dos elementos importantes para definir el comportamiento de nuestra `Toolbar`, los dos son propiedades del elemento `CollapsingToolbarLayout`.

```
app:layout_scrollFlags="scroll|exitUntilCollapsed"
```

- `exitUntilCollapse`: Este flag hace que el `AppBar` realice un `scrollOff` hasta que llegue al tamaño del `toolbar`, ahí se detendrá y dejara de hacer `scroll` quedando solamente nuestro `toolbar`.
- `scroll`: Este flag debe estar en todas las vistas que van a hacer un "offScreen", las que no contengan este flag, se mantendrán en la parte superior de la pantalla.



```
app:contentScrim="?attr/colorPrimary"
```

Y que define el color de la AppBar cuando finalice el scroll.

En la configuración del ImageView `app:layout_collapseMode="parallax"` indica que cuando el toolbar se colapse, la imagen lo haga en modo parallax, es decir que lo haga a una velocidad diferente para darle una animación más elegante.

Para declarar el contenido que estará debajo de la Toolbar es obligatorio definirlo dentro de un contenedor especial llamado `NestedScrollView`, tal y como hicimos anteriormente:

```
> </android.support.design.widget.AppBarLayout>

> <android.support.v4.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:text="Con diez cañones por banda, viento en popa a toda vela,..."
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textStyle="bold"
        android:textSize="18dp"
        app:layout_behavior="android.support.design.widget.AppBarLayout$ScrollingView..." />
    </android.support.v4.widget.NestedScrollView>
> </android.support.design.widget.CoordinatorLayout>
```

## ¿CÓMO INCLUIR TOOLBAR EN OTRAS PARTES DE LA INTERFAZ?

Otra ventaja que podemos aprovechar al disponer del control Toolbar como componente independiente es que podemos utilizarlo en otros lugares de nuestra interfaz, y no siempre como barra de acciones superior.

Así, podríamos por ejemplo utilizar un componente toolbar dentro de una tarjeta. Para ello, añadamos una tarjeta a nuestra aplicación de ejemplo, y simplemente incluyamos en su interior un control Toolbar de la misma forma que hemos hecho antes (`activity_main_v7`):



```

} <android.support.v4.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:orientation="vertical" >

        <android.support.v7.widget.CardView
            xmlns:app="http://schemas.android.com/apk/res-auto"
            android:id="@+id/card_view"
            android:layout_gravity="center"
            android:layout_width="match_parent"
            android:layout_height="200dp"
            app:cardUseCompatPadding="true"
            app:cardCornerRadius="4dp">

            <android.support.v7.widget.Toolbar
                android:id="@+id/TbCard"
                android:layout_height="?attr/actionBarSize"
                android:layout_width="match_parent"
                android:minHeight="?attr/actionBarSize"
                android:theme="@style/ThemeOverlay.AppCompat.ActionBar"
                app:popupTheme="@style/ThemeOverlay.AppCompat.Light" >

            </android.support.v7.widget.Toolbar>
        </android.support.v7.widget.CardView>
    </LinearLayout>
} </android.support.v4.widget.NestedScrollView>

```

Si ejecutáramos la aplicación en este momento, la toolbar no se vería ya que no le hemos asignado ningún título ni ningún menú. Anteriormente no tuvimos que hacer esto de forma explícita porque al indicar que la toolbar iba a hacer las función de *app bar* (mediante la llamada a `setSupportActionBar()`), el título y el menú lo tomó automáticamente de la actividad asociada. Sin embargo, en esta ocasión la toolbar es independiente de la actividad, por lo que tendremos que asignar estos elementos nosotros mismos. Por ejemplo si queremos asignar en el xml el título lo haremos con la etiqueta `app:title`, aunque también podemos realizarlo mediante código. Para ello, desde el método `onCreate()` de la actividad recuperaremos una referencia al control, y llamaremos a sus métodos `setTitle()` e `inflateMenu()` para asignar el título y el menú respectivamente.

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main_v7);

    toolbar=(Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    ActionBar actionBar=getSupportActionBar();

    actionBar.setHomeAsUpIndicator(R.drawable.ic_menu);
    actionBar.setDisplayHomeAsUpEnabled(true);

    Toolbar tbCard = (Toolbar) findViewById(R.id.TbCard);
    tbCard.setTitle("Mi tarjeta");

    tbCard.setOnMenuItemClickListener(
        new Toolbar.OnMenuItemClickListener() {
            @Override
            public boolean onMenuItemClick(MenuItem item) {

                switch (item.getItemId()) {
                    case R.id.action_1:
                        Log.i("Toolbar 2", "Acción Tarjeta 1");
                        break;
                    case R.id.action_2:
                        Log.i("Toolbar 2", "Acción Tarjeta 2");
                        break;
                }

                return true;
            }
        });

    tbCard.inflateMenu(R.menu.menu_tarjeta);
}

```

Para mi caso de ejemplo he definido un nuevo menú menu\_tarjeta (definido en el fichero /res/menu/menu\_tarjeta.xml) con dos acciones de muestra:

```

<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools" tools:context=".MainActivity">
    <item android:id="@+id/action_1" android:title="Op1Menu cardView"
        android:orderInCategory="100" app:showAsAction="never" />
    <item android:id="@+id/action_2" android:title="Op1Menu cardView"
        android:orderInCategory="100" app:showAsAction="never" />
</menu>

```

